Events and Asynchronous Notification

Events, Observables, Pub/sub, and other patterns

December 19, 2017

Contents

- Events vs. Property Notifications
- Asynchronous Notification patterns
- Conditional Notification patterns
- Created item pattern for Events
- Actions using created items

Event vs. Property State Change

- Events may be used when a simple property state change is not sufficient
- Examples: shortpress, longpress, button held
- Also for significant state changes that follow a common pattern, e.g. high abnormal reading
- Events emit outputData representations
- Event Notifications may reuse the same patterns as property state change Notifications

What should the interaction abstraction for Events be?

- A location, like a property or action, which emits representations of state changes
- OutputData describes the message payload or other representation, with data fields
- The Event Interaction Form may describe an operation where an observable item is created, or it may point to an existing observable item
- Event may optionally offer a getEvents form that allows batch retrieval of events

Notification Patterns (Events and Properties)

- CoAP Observe, extended GET
- HTTP EventSource
- HTTP Long Poll
- MQTT Publish/Subscribe
- Websockets
- Web Hook
- Dynamic Link to another resource
- Monitor resource that can be observed or retrieved

Created Resource pattern

- Sometimes notifications are received by observing a static or pre-existing resource or item
- Sometimes notifications are received by creating a resource or item, then interacting with the created item
  - New MQTT Topic,
  - websocket or Event Source tag,
  - monitior resource to observe,
  - web hook or dynamic link

How should the binding for events work?

- Same as the binding for observing properties
- Same set of methods for receiving notifications

Notification binding for CoAP Observe

- Static resource is the common pattern
- Existing pattern covers CoAP Observe

Notification Binding for Pub/Sub

- Static pattern using MQTT vocabulary as per the current examples
- Dynamic topic creation pattern should result in the same subscription operation as the static pattern
- Client should not need to adapt at the application level, just use the observe form whether the topic is static or created

Notification Binding for websockets

- Created item is the most likely pattern, with reuse of a port for several observe relationships
- Client should be able to invoke observe and receive notifications as in the static resource pattern, transparent to the application

Notification binding for Web Hooks and Dynamic Links

- Created item is the only option
- Does not return an observable
- Pushes notifications to a client-specified location
- This is a different pattern for the client to understand
- Client may be able to observe the destination location (may be local to the client) as a separate operation
- Client role as third party in orchestration

Conditional Notification Binding

- Sometimes conditional settings are input as request parameters
- Sometimes a resource configuration; applies to all requests
- Can also be attributes of a web hook or dynlink
- Abstract part of an interaction supplies the parameters to be passed to the protocol binding
- TD may describe "set conditional parameters" as a separate interaction or included parameters in an observe interaction

Server/Broker reuse

- Pub/Sub and websockets may re-use a base address for many observe relationships
- Setup/Initialization/Connection phase before clients can use observe/subscribe
- TD extensions to the base item could facilitate this (see issue #14)

Bindings for created items

- outputData is generated by the created item
- The location of the created item is returned when the observe form is processed
- A representation of the created item may be returned in the response payload of the observe
- There can be forms pre-loaded into the TD that describe the operations on the created items
- The forms for created items would have href place holders that are filled in from the create operation response data

Example form for dynamically creating an item

```
"form": [
 {
  "href": "/example/event",
  "rel" : ["observe"],
  "observeMethod": "create",
  "http:methodName": "http:post"
  "http:responseHeader": [
   {
    "http:fieldName": "Location",
    "http:fieldValue": { "@type": "td:uriLocation" }
   }
  ]
 } //(continued)
```

Interacting with created items

```
{
 "href":{"@type": "td:uriLocation"},
 "rel": ["observeCancel"],
 "http:methodName": "http:delete"
},
 {
 "href":{"@type": "td:uriLocation"},
 "rel": ["getEvents"],
 "http:methodName": "http:get"
},
{
 "href":{"@type": "td:uriLocation"},
 "rel": ["updateEvents"],
 "http:methodName": "http:put"
 }
]
```

Dynamically created actions

- Outputdata constructs can describe the payload and data constraints
- output links (rel=actionStatus, rel=actionCancel, rel=actionUpdate) can describe the messages
- The href will be unknown until the Action is invoked
- A form entry that has a variable for the href field would be a potential design direction
- The variable would be filled in using the returned location pointer after the action is invoked
- The invokeAction form could specify the create pattern using a relation type

Example Form for created action

```
"form": [
 {
  "href": "/example/actions",
  "rel" : ["invokeAction"],
  "invokeMethod": "createAction",
  "http:methodName": "http:post"
 },
 {
  "href":{"@type": "td:uriLocation"},
  "rel": ["actionCancel"],
  "http:methodName": "http:delete"
 },
 {
  "href":{"@type": "td:uriLocation"},
  "rel": ["actionStatus"],
  "http:methodName": "http:get"
 },
 {
  "href":{"@type": "td:uriLocation"},
  "rel": ["actionUpdate"],
  "http:methodName": "http:put"
  }
]
```

Patterns can be combined

- Initial status of the action can be returned in the response payload when the Action is invoked and the new resource is created
- The location of the created resource may be returned in the initial response payload, or in a header field, or both
- The response to Action Invoke may be itself be an observable and return asynchronous status updates in response to the Action invocation

Describe the response header in the transport vocabulary

```
"form": [
 {
  "href": "/example/actions",
  "rel" : ["invokeAction"],
  "invokeMethod": "create",
  "http:methodName": "http:post"
  "http:responseHeader": [
   {
    "http:fieldName": "Location",
    "http:fieldValue": { "@type": "td:uriLocation" }
   }
  ]
 }
]
```

Location in the response payload

```
"outputData": {
 "type": "object",
 "field": [
  {
   "name": "id",
   "value": { "@type": "td:uriLocation" }
  },
  {
    "name": "currentStatus",
    "value": { "@type": "td:actionStatus" }
  }
  {
    "name": "createdAt",
    "value": { "@type": "td:actionInvokeTime" }
  }
 ]
```